Chapter 1

Parallel solution of a 2D Poisson problem with flux boundary conditions

This document provides an overview of how to distribute the 2D Poisson problem with flux boundary conditions. It is part of a series of tutorials that discuss how to modify existing serial driver codes so that the Problem object can be distributed across multiple processors.

A feature of this problem is that the flux boundary conditions are applied by attaching "flux elements" (derived from the FaceElement base class) to the "bulk elements" adjacent to the appropriate mesh boundary. As discussed in the tutorial for the serial driver code, the FaceElements are not involved in any adaptation within the bulk mesh. Instead, they are detached before the bulk mesh is adapted and re-attached afterwards, which ensures that the FaceElements are only attached to bulk elements present in the adapted mesh.

The same issue arises when the Problem is distributed: all FaceElements must be attached before the problem is distributed to allow METIS to analyse the interaction between face and bulk elements correctly. However, after the Problem has been distributed, some of the bulk elements on each processor will have been deleted, leaving the corresponding FaceElement dangling. To deal with such problems, oomph-lib provides the empty virtual functions
`Problem::actions_before_distribute()`

and
`Problem::actions_after_distribute()`

which are called automatically by `Problem::distribute(...)`. Specifically, `Problem::actions_before_distribute()` is called **after** the problem distribution has been determined by METIS but **before** the actual distribution (during which elements are deleted) takes place. `Problem::actions_after_distribute()` is called after the problem distribution is complete.

In the present problem we overload the functions `Problem::actions_before_distribute()` and `Problem::actions_after_distribute()` to perform the same functions as `actions_before_adapt()` (i.e. delete the flux elements) and `actions_after_adapt()` (i.e. re-attach the flux elements).
We note that any FaceElement that is attached to a halo element in the bulk mesh becomes a halo element itself; see the general MPI tutorial for further details.

Most of driver code is identical to its serial counterpart and we only discuss the changes required to distribute the problem. Please refer to another tutorial for a more detailed discussion of the problem and its (serial) implementation.

1.1 The main function

The only changes required to the main function are the usual calls to initialise and finalise oomph-lib's MPI routines and a single call to `Problem::distribute()` after the problem has been constructed. The source code is actually slightly more complicated because the distribution is read in from a file so that the driver can be used as a self-test. Note that the file must specify the partition for **all** elements, including the FaceElements. (We refer to another tutorial for details on how to create the distribution file.)

1.2 The problem class

The only additions to the problem class are the functions `actions_before_distribute()` and `actions_after_distribute()`. As explained above, these perform exactly the same functions as `actions_before_adapt()` and `actions_after_adapt()`, respectively.

```
 /// Actions before distribute: Wipe the mesh of prescribed flux elements
 /// (simply call actions_before_adapt() which does the same thing)
 void actions_before_distribute()
  {
   actions_before_adapt();
  }
 /// Actions after distribute: Rebuild the mesh of prescribed flux
 /// elements (simply call actions_after_adapt() which does the same thing)
 void actions_after_distribute()
  {
   actions_after_adapt();
  }
```

1.3 The doc_solution() function

As with other driver codes, the output files are modified to allow each processor to output its elements into files that include the processor number.

```
//======================start_of_doc=======================================
/// Doc the solution: doc_info contains labels/output directory etc.
//======================================================================
template<class ELEMENT>
void RefineableTwoMeshFluxPoissonProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{
 // Doc refinement levels in bulk mesh
 unsigned min_refinement_level;
 unsigned max_refinement_level;
 Bulk_mesh_pt->get_refinement_levels(min_refinement_level,
                                     max_refinement_level);
 cout << "Ultimate min/max. refinement levels in bulk mesh : "
      << min_refinement_level << " "
      << max_refinement_level << std::endl;
 ofstream some_file;
 char filename[100];
 // Number of plot points
 unsigned npts;
 npts=5;
 // Output solution with halo elements
 //-----------------------------------
 Bulk_mesh_pt->enable_output_of_halo_elements();
 sprintf(filename,"%s/soln_with_halo%i_on_proc%i.dat",
         doc_info.directory().c_str(),
         doc_info.number(),this->communicator_pt()->my_rank());
 some_file.open(filename);
 Bulk_mesh_pt->output(some_file,npts);
 some_file.close();
 Bulk_mesh_pt->disable_output_of_halo_elements();
 // Output solution
 //-----------------
 sprintf(filename,"%s/soln%i_on_proc%i.dat",doc_info.directory().c_str(),
         doc_info.number(),this->communicator_pt()->my_rank());
 some_file.open(filename);
 Bulk_mesh_pt->output(some_file,npts);
 some_file.close();
 // Output exact solution
 //----------------------
 sprintf(filename,"%s/exact_soln%i_on_proc%i.dat",doc_info.directory().c_str(),
         doc_info.number(),this->communicator_pt()->my_rank());
 some_file.open(filename);
 Bulk_mesh_pt->output_fct(some_file,npts,TanhSolnForPoisson::get_exact_u);
 some_file.close();
 // Doc error and return of the square of the L2 error
 //---------------------------------------------------
 double error,norm;
 sprintf(filename,"%s/error%i_on_proc%i.dat",doc_info.directory().c_str(),
         doc_info.number(),this->communicator_pt()->my_rank());
 some_file.open(filename);
 Bulk_mesh_pt->compute_error(some_file,TanhSolnForPoisson::get_exact_u,
                             error,norm);
 some_file.close();
 // Doc L2 error and norm of solution
 cout << "\nNorm of error   : " << sqrt(error) << std::endl;
 cout << "Norm of solution: " << sqrt(norm) << std::endl << std::endl;
} // end of doc
```

The remainder of this driver code is unchanged from the `serial version`.

1.4 Source files for this tutorial

- The source files for this tutorial are located in the directory:

  demo_drivers/mpi/distribution/two_d_poisson_flux_bc_adapt/

- The driver code is:

  demo_drivers/mpi/distribution/two_d_poisson_flux_bc_adapt/two_d_poisson_flux_bc_adapt.cc

1.5 PDF file

A pdf version of this document is available.